Object-Oriented Layers in ELIST

Mary Ann Widing, Kathy Lee Simunich, Dariusz Blachowicz, Mary Braun, and Dr. Charles Van Groningen
Argonne National Laboratory

In developing large, complex software systems, object-oriented programming techniques can provide many benefits. In addition to using an object-oriented language, developers should also employ other techniques such as layers to fully obtain these benefits. This article discusses several of these design details that were used in developing a military logistics system called Enhanced Logistics Intra-Theater Support Tool.

Planning for the transportation of large amounts of equipment, troops, and supplies presents a complex problem for military analysts. Software tools are critical in defining and analyzing these plans. Argonne National Laboratory developed the Enhanced Logistics Intra-Theater Support Tool (ELIST) to assist military planners in determining the logistical feasibility of an intra-theater course of action. This article focuses on the object-oriented design strategies used in developing the latest version of this system. Details of the specific military, logistical algorithms that were implemented can be found in other sources [1].

ELIST Model Requirements

The military logistics community has successfully used the previous version of ELIST (v.7) in planning analyses and training exercises for a number of years [2]. Ongoing use of this system has led to requests for more detail, more capabilities, and increased flexibility. Users wanted to model the transportation of military cargo at the individual vehicle level with a much more detailed simulation than in the existing ELIST system. Because of the size and complexity of the new logistics transportation model, performance was also a primary consideration. ELIST needed to be more reliable with a more robust data storage and handling system to address increased data requirements. Therefore, in developing this new version, Argonne National Laboratory took advantage of the opportunity to perform a total redesign of the program architecture.

Multiple languages were used to implement the previous version of ELIST. Initially, Prolog was used for most of the data and model representations and computations. C components and libraries were used for computations, user interface, and integration. Although ELIST was a very successful application, this multi-language approach proved difficult and time-consuming to port and maintain.

For the new ELIST, the Java language was selected for many reasons. Java supports object-oriented features such as encapsulation, inheritance, abstraction, and polymorphism. Using Java would solve many portability concerns because of the availability of Java virtual machines on multiple platforms. The standard Java developer's kit provides built-in packages for user interface, database-connectivity, and distributed processing that address many maintenance concerns. Java's memory management and exception handling schemes address reliability concerns. Oracle was chosen as the database management system for the new version of ELIST because it would address many data storage requirements and was already in use at sponsor's sites.

Object-Oriented Design Approach

We chose evolutionary delivery for our lifecycle model [3]. Under this approach, we developed the new version of ELIST, showed it to users, and refined the software based on their feedback. The first step was to specify all of the logistical algorithms in a requirements document based on knowledge gained during prior model development and from interaction with the user community.

Based on these algorithms, we created Unified Modeling Language (UML) diagrams of the basic simulation objects. Using these requirements, we put our initial emphasis on developing the visual aspects of the system needed to support the data required by the simulation. As full functionality was added to these areas, it became apparent that more than a thousand classes would be required in the complete system.

In structuring an application of this complexity, we needed to employ a scheme for partitioning the software into manageable sections. We chose to use class-type architecture for our design [4]. In class-type architecture, the classes of the application are organized into well-defined layers based on their general function. Figure 1 shows the overall architecture of the ELIST system.

Each layer is well modularized and addresses a specific area of responsibility. The different layers can be developed relatively independently with an interface specifying their use by other layers. In designing each of these layers, we followed the recursive/parallel model [5], dividing each layer into subcomponents and gradually refining the classes as development progressed.

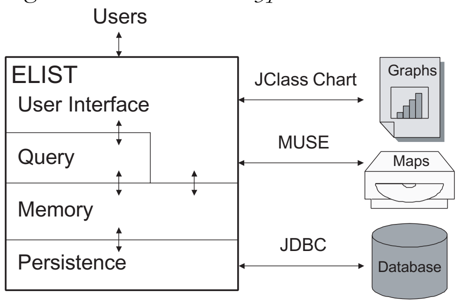
This design approach has many advantages. Changes to one layer are isolated from other layers, making the application more portable, extensible, and maintainable. In addition, different software teams can concentrate on different layers, drawing on their areas of expertise. Many of these independent layers can be structured as general-purpose packages in a code repository used across multiple projects. This approach enabled us to leverage the development efforts across multiple projects, saving expense and increasing code reliability.

ELIST is composed of four main layers: the user interface layer, query layer, memory layer, and persistence layer. In each layer, UML was used to define classes and the relationships among them. Each layer presented its own set of issues that needed to be addressed in organizing the classes. In the sections that follow, each layer is discussed in detail, focusing on some of the techniques used in that layer.

User Interface Layer

The topmost layer of the ELIST application is the user interface layer. Written using the Java Foundation Classes, this

Figure 1: *ELIST Class-Type Architecture*



layer presents graphical windows to the user. In developing the window designs, we prototyped windows and presented them to the user community for iterative feedback before actual code development began.

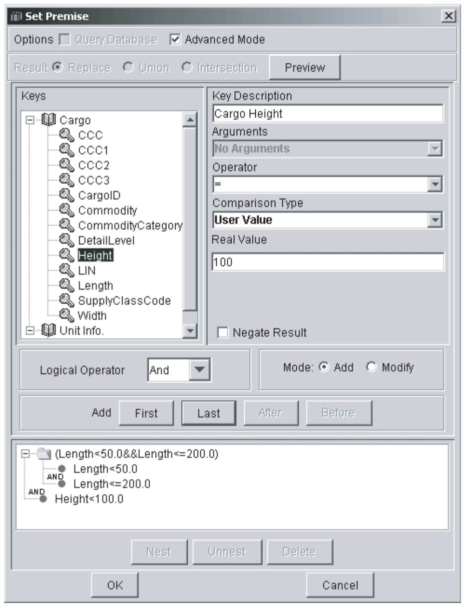
ELIST requires both traditional widgets such as tables, as well as custom widgets such as specialized trees and Gantt charts. An extensive package of generic, user interface widgets was developed for several reasons. One is that the standard Java widgets contain a large number of bugs. By developing our own widgets that map to these standard widgets, we were able to provide the bug fixes that were required as well as add custom features to the widgets. As new versions of Java are released, we will update only the user interface layer to accommodate any changes; this greatly increases maintainability of our models.

The commercial tool called JClass Chart from Sitraka was accessed to create standard graphs using a package within the user interface layer. Again, this allows us to switch tools if needed and add functionality beyond that supplied in the tools.

Most of our geographical information system (GIS) requirements could be implemented by writing a package that uses the 2D graphics package provided in the standard Java system. However, to display images created from standard map products, we wrote an interface on top of National Imagery and Mapping Agency (NIMA) Mapping, Charting, and Geodesy, Utility Software Environment (MUSE) software using the Java Native Interface utility.

The MUSE library provides routines for reading and writing standard NIMA map products. This gives us the flexibility to completely integrate our map windows with other parts of our application while taking advantage of existing code for reading the map products. In implementing this package, we used the technique called *wrapping*. Object-oriented classes were written to interface to non-object-oriented functions within a library.

Figure 2: *Query Window*


Set Premise

Options [ ] Query Database [✓ Advanced Mode]

Result [✓ Replace] Union Intersection

Key Description: Cargo Height

Arguments: No Arguments

Operator: =

Comparison Type: User Value

Real Value: 100

[ ] Negate Result

Logical Operator: And

Mode: [✓ Add] Modify

```
(Length<50.0&&Length<=200.0)
    Length<50.0
AND
    Length<=200.0
AND
Height<100.0
```

A main editing window was available in the interface for each of the main objects in the memory layer. Each of these top-level windows organized the data for that object and provided multiple, related tabbed panels of information.

Query Layer

When dealing with huge amounts of data, users need a dynamic, flexible mechanism for retrieving subsets for various types of processing such as viewing, modifying, or tallying results. We developed the query layer to provide users with a way to build, save, retrieve, and execute complex queries about their data. When executed, each query returns collections of objects that match a defined premise.

The query package provides generic query and data assignment capability. Any object that is to be queried must publish what information can be retrieved or modified, and what data values are valid by implementing the *QueryObject* and *QueryObjectSummary* interfaces. The query system does not need to know any other information about the structure or function of the objects.

We designed the query package in three sub-layers: user interface, logical operations, and data management. The query package dynamically creates a window that allows users to construct queries and data assignments based on the information published by the data objects. Through user interface windows, users build arbitrarily complex expressions by nesting simple predicate expressions in a tree-like structure, as shown in Figure 2.

When the user is creating this tree in the interface, the system builds a corresponding hierarchy of *PredicateExpression* classes and *ConditionalStatement* classes in the logical operations layer. The first allows the construction of arbitrarily complex expressions while the second allows modification of data values within an object.

After the logical operations classes are created, the data support and management layer performs the query on the set of data objects. These data objects are typically in memory but may optionally be in a relational database. In this case, the query package can retrieve and store data in a relational database via Structured Query Language (SQL) statements using query keys that have been mapped to database fields. The predicate expression generates the *where* clause of a SQL statement, which is then sent to the *PersistenceBroker*, which in turn builds the complete SQL statement and executes it.

Memory Layer

The heart of ELIST is its simulation, so in designing its memory layer (or business layer), the simulation's requirements were our primary concern. In examining the data requirements, we found that data can be divided into a number of main objects that have dependencies on other objects. Figure 3 shows the main objects in ELIST's memory layer.

These main objects represent logical divisions in the data. The user interface was structured to correspond to this division of objects by creating one main editing window for each of these objects. As shown by the arrows in Figure 3, each main object may depend on other objects. To support handling these dependencies, a Java interface *DependentObject* was defined. Each main object implemented this interface. By redefining basic methods in the interface, each object specified which other objects it depended on. This gave us a scheme for easily checking which objects were affected by changes in other objects. For example, if a user wants to edit a new network, we could quickly determine that any currently loaded scenarios would have to be unloaded. This enabled us to keep the object dependencies in the memory layer rather than hard-coding it in the user interface.

Metadata for each of these main objects were mapped to corresponding database tables that could be managed through tables in the user interface. Important metadata included descriptions, modification dates, owners, and classification levels. Including these data in our design enabled users to more easily track changes being made for different strategic plans.

Whenever objects are edited in the user interface windows, the corresponding objects are immediately changed in the memory layer, but not in the database. To support this feature, classes were developed that implement a *ChangeLog*. When a text field or other widget is edited, the corresponding memory layer objects are changed and a change record is created. All changes, whether updates, adds, or deletes, are stored in a queue associated with the window. When the user explicitly requests a save, this log is then used to propagate the updates to the database through the persistence layer. Special group records allow a set of changes to be grouped together. The user can display an undo log at any time and may roll back changes in memory.

Figure 4 shows a UML diagram containing the main *ChangeLog* classes. As objects are edited in the user interface, methods in the *ChangeLog* class create instances of the appropriate type of *ChangeRecord* object.

Persistence Layer

Proper object-relational integration requires a strategy for mapping the object model to the relational model in order for Java objects to become persistent (saved for later use) in a relational database management system (RDBMS). Without some strategy, objects cannot be directly saved to and retrieved from relational databases. This problem of trying to maintain consistency between the objects in memory and the state of the database leads to writing hundreds of lines of embedded SQL code for reading and writing to the database.

There is a standard package available in Java for interfacing with commercial relational databases called Java Data Base Connectivity (JDBC). This package allows applications to connect to a wide variety of database products in a standard way. However, JDBC is still a lower-level application programming interface that does not facilitate a nice, modular encapsulation of the mapping needed to make memory layer objects persistent. To fully support our class-type architecture, we implemented a persistence layer that wraps the lower-level functionality of JDBC [6]. This provides a means for the objects in memory to create, retrieve, update, and delete themselves in the database. Figure 5 shows the main classes defined in the persistence layer.

Every object that needs to be persistent is a subclass of *PersistentObject*. The *ClassMap* class is defined to map an object to a table in the relational database. It separates the persistence mechanism from the object schema. In implementing these objects, a standard was adopted in which a subdirectory called *classmap* was defined under each package directory containing *PersistentObjects*. The corresponding *ClassMap* classes for those objects were stored in that subdirectory. For every type of *PersistentObject*, a *ClassMap* instance is created that stores the information needed to create *SELECT*, *INSERT*, *UPDATE*, and *DELETE* SQL statements and records information on the database table and columns used. The *ClassMap* object implements the database access for the corresponding *PersistentObject*.

The main class in the persistence layer is the *PersistenceBroker* class. This object acts as the database manager for ELIST, maintaining the connection to the RDBMS. It handles communication between objects in the application and the persistence mechanism by wrapping the actual calls to JDBC. The *PersistenceBroker* holds the collection of *ClassMaps* for all *PersistentObjects* in memory. By using calls to JDBC, the *PersistenceBroker* class implements *saveObject*, *retrieveObject*, and *deleteObject* methods. It also implements a *processSQL* method that can submit any arbitrary SQL call.

When the user is editing data, the persistence layer works in conjunction with the *ChangeLog* mechanism. When a user selects a save option from an editing window, the *ChangeLog* for that window is used to forward those saves to the appropriate *PersistentObjects*. The *PersistenceBroker* finds the corresponding *ClassMap* for that class of *PersistentObject* and calls it to construct the appropriate SQL statement for the object. It then attempts to process the SQL statement using JDBC. If there are errors, the database is rolled back, a *PersistenceException* is thrown to the user interface layer; otherwise, the transaction was successful and the changes to the database are committed. Through the use of this exception handling mechanism, we were able to keep the persistence layer sep-

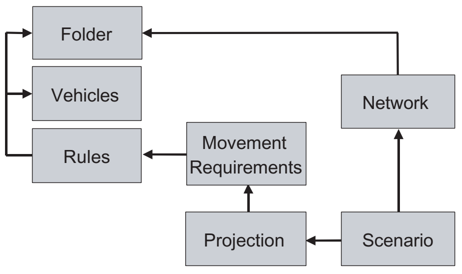


Figure 3: *Dependencies of Main Objects*

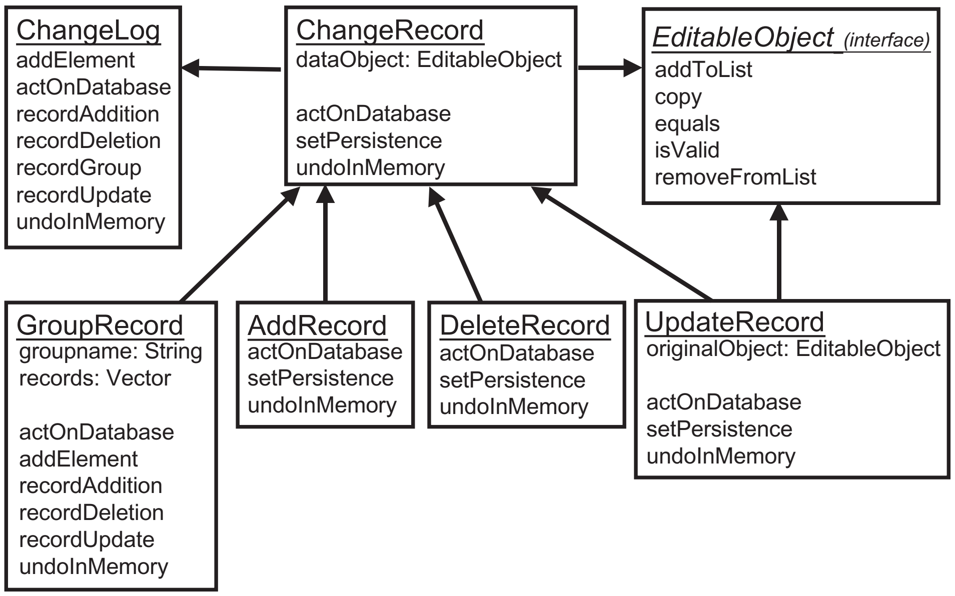
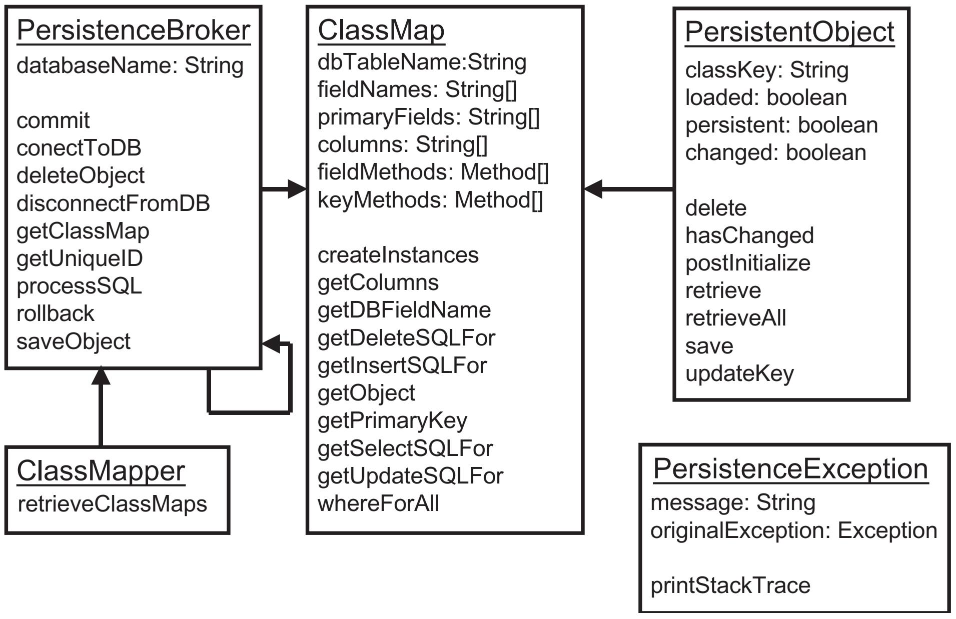
Figure 4: *ChangeLog Classes*


Figure 5: *Persistence Layer Classes*


arated from the user interface layer, and at the same time, keep the database in sync with the objects in memory.

Summary

Through ELIST development, we learned that it is essential to apply object-oriented techniques throughout many levels of our design. In addition to using an object-oriented language, we structured our application using class-type architecture. By dividing our application into layers, we were able to focus on separate, reusable components and assign lead developers to each layer who specialized in the respective component areas. By carefully designing each layer using UML modeling techniques, we addressed our primary concerns regarding portability, maintainability, and reusability. The resulting ELIST system has been successfully delivered to the sponsor and is evolving in response to new and refined requirements. The packages developed to support the various layers have been reused on multiple government projects, providing substantial cost savings for those development efforts as well.◆

Acknowledgment

This work was supported under a military interdepartmental purchase request from the U.S. Department of Defense, Military Traffic Management Command Transportation Engineering Agency, through the U.S. Department of Energy contract W-31-109-ENG-109.

References

1. Braun, M.D., and C. Van Groningen. ELIST 8 Transportation Model. ANL/DIS/02-1. Argonne, IL: Argonne National Laboratory, 13 Feb. 2002.
2. Macal, C., C. Van Groningen, and M. Braun. Simulation of Transportation Movements Over Constrained Infrastructure Networks. Proc. of the 1995 Simulation Multi-Conference, Phoenix, AZ, 27 Apr. 1995 (4): 97-102.
3. McConnell, Steve. Rapid Development: Taming Wild Software Schedules. Redmond, WA: Microsoft Press, 1996.
4. Ambler, Scott W. Building Object Applications That Work: Your Step-by-Step Handbook for Developing Robust Systems with Object Technology. New York: Cambridge University Press, 1998.
5. Berard, Edward V. "Understanding the Recursive/Parallel Life-Cycle." Hotline of Object-Oriented Technology 1.7 (May) 1990: 10-13.
6. Ambler, Scott W. "Mapping Objects to Relational Databases." White Paper. AmbySoft Inc., 26 Feb. 1999.

About the Authors

Mary Ann Widing is an information systems engineer in the Decision and Information Sciences Division at Argonne National Laboratory. Her work at Argonne has focused on developing complex, graphical user interfaces for decision support systems used by government agencies. Widing has a Bachelor of Science and Master of Science in engineering from the University of Illinois in Urbana-Champaign.

Argonne National Laboratory
9700 South Cass Ave.
Argonne, IL 60439-4832
Phone: (630) 252-3798
Fax: (630) 252-6073
E-mail: widing@dis.anl.gov

Kathy Lee Simunich is a computer scientist in the Decision and Information Sciences Division at Argonne National Laboratory. Her work at Argonne includes environmental modeling and object-to-relational databases, as well as writing reusable components across various Department of Defense and Department of Energy projects. Simunich has a Bachelor of Science in meteorology from Northern Illinois University and a Master of Science in computer science from North Central College in Illinois.

Argonne National Laboratory
9700 South Cass Ave.
Argonne, IL 60439-4832
Phone: (630) 252-3285
Fax: (630) 252-6073
E-mail: simunich@dis.anl.gov

Dariusz Blachowicz is a computer scientist in the Decision and Information Science Division at Argonne National Laboratory. His work at Argonne includes a wide range of modeling and simulation applications, and Web-based interactive applications for Department of Defense and Department of Energy agencies. Blachowicz has a Bachelor of Science in civil engineering and a Master of Science in computer science from Illinois Institute of Technology in Chicago.

Argonne National Laboratory
9700 South Cass Ave.
Argonne, IL 60439-4832
Phone: (630) 252-6187
Fax: (630) 252-6073
E-mail: blach@dis.anl.gov

Mary Braun is a computer systems engineer in the Decision and Information Sciences Division at Argonne National Laboratory. Her work has focused on military logistics modeling and simulation. Braun has a Bachelor of Science from the University of Santa Clara and a Master of Science from the University of California, Berkeley, both in electrical engineering.

Argonne National Laboratory
9700 South Cass Ave.
Argonne, IL 60439-4832
Phone: (630) 252-3727
Fax: (630) 252-6073
E-mail: duffy@dis.anl.gov

Charles Van Groningen, Ph.D., leads the Enhanced Logistics Intra-Theater Support Tool development team at Argonne National Laboratory. His research interests include modeling, simulation, and knowledge representation. Van Groningen has a doctorate in artificial intelligence from the Illinois Institute of Technology, a Master of Science in computer science from DePaul University, and a Bachelor of Arts in mathematics from Trinity Christian College.

Argonne National Laboratory
9700 South Cass Ave.
Argonne, IL 60439-4832
Phone: (630) 252-5308
Fax: (630) 252-6073
E-mail: vang@dis.anl.gov